공학석사학위논문

# 웹어셈블리를 활용하는 어플리케이션을 위한 스냅샷 기반 연산 오프로딩

Snapshot-based Computation Offloading for Web Application Using WebAssembly

2019년 8월

서울대학교 대학원

전기 컴퓨터 공학부

신 창 현

공학석사학위논문

# 웹어셈블리를 활용하는 어플리케이션을 위한 스냅샷 기반 연산 오프로딩

Snapshot-based Computation Offloading for Web Application Using WebAssembly

2019년 8월

서울대학교 대학원

전기 컴퓨터 공학부

신 창 현

# 웹어셈블리를 활용하는 어플리케이션을 위한 스냅샷 기반 연산 오프로딩

Snapshot-based Computation Offloading for Web Application Using WebAssembly

지도교수 문 수 묵

이 논문을 공학석사 학위논문으로 제출함

2019년 8월

서울대학교 대학원

전기 컴퓨터 공학부

신 창 현

신창현의 공학석사 학위 논문을 인준함

2019년 8월

| | | |
|---|---|---|
| 위 원 장: | 백 윤 흥 | (인) |
| 부위원장: | 문 수 묵 | (인) |
| 위 원: | 윤 성 로 | (인) |

# 초록

웹 기술이 발전하면서 웹 브라우저와 Node.js가 설치된 환경이라면 어디서든 웹 어플리케이션을 실행할 수 있게 되었다. 그러나 하드웨어 성능이 부족한 기기에서 많은 연산량을 요구하는 웹 어플리케이션을 실행하는 것은 어려운 문제이다. 연산 오프로딩은 이러한 문제를 해결하기 위한 방법 중 하나로 어플리케이션 자원이 제한된 클라이언트 성능이 좋은 서버가 연산을 대신 수행하고 결과를 돌려주는 방식이다. 서버에서 클라이언트에서 수행중인 작업을 이어서 하기 위해서는 실행 상태를 저장하고 복원하는 작업이 필요하다. 스냅샷 기반 연산 오프로딩[2]은 웹 어플리케이션의 상태를 스냅샷[10]을 통해 저장하고 복원함으로써 연산 오프로딩을 수행하는 기술이다. 그러나 기존의 스냅샷은 웹어셈블리[3]라는 새로운 기술을 사용하는 어플리케이션에 대해서는 적용할 수 없다는 문제가 있었다.

웹어셈블리는 웹 표준 언어인 자바스크립트의 부족한 성능을 보완하기 위해 제안된 새로운 형식의 언어이다. 자바스크립트와는 다른 방식의 로딩으로 인해 기존의 스냅샷 방식은 웹어셈블리의 상태를 저장하지 못해 스냅샷 기반의 연산 오프로딩을 적용할 수 없었다.

본 논문에서는 웹 어플리케이션의 실행 상태를 저장하고 복원할 수 있는 새로운 스냅샷 기반 연산 오프로딩 방식을 제안한다. 우리는 웹어셈블리의 상태를 복원하기 위한 코드를 생성를 생성하고 캡처한 메모리를 캡처하여 문제를 해결하였다. 우리는 웹어셈블리를 사용하는 어플리케이션에 대해 새로운 방식을 적용하고 평가하였다. 평가 결과 스냅샷 기반의 연산 오프로딩이 짧은 마이그레이션 시간과 실행 성능 향상을 보여주는 것을 확인할 수 있었다.

**주요어**: 웹어셈블리, 오프로딩, 스냅샷, 자바스크립트, 웹

**학 번**: 2017-28910

# 목차

**초록** **i**

**제 1 장 서론** **1**

**제 2 장 배경지식** **3**

2.1 웹어셈블리 . . . . . 3

2.2 스냅샷 기반 연산 오프로딩 . . . . . 5

2.3 모바일 웹 워커 . . . . . 7

**제 3 장 웹어셈블리 상태 저장 및 복원** **10**

3.1 웹어셈블리 함수의 복원 . . . . . 11

3.2 웹어셈블리 메모리의 복원 . . . . . 14

**제 4 장 실험 및 결과** **16**

4.1 실험 환경 . . . . . 16

4.2 스냅샷 기반의 상태 저장 및 복원 성능 . . . . . 17

4.3 실행 성능 . . . . . 18

**제 5 장 결론** **21**

# 표 목차

표 1 큐브 개수에 따른 스냅샷 캡처 시간 및 스냅샷과 웹어셈블리 메모리의 크기 . . . . . 18

# 그림 목차

그림 1 웹어셈블리의 컴파일 과정 . . . . . 4
그림 2 웹 어플리케이션에서의 웹어셈블리 초기화 과정 . . . . . 5
그림 3 스냅샷 기반 연산 오프로딩의 실행 과정 . . . . . 6
그림 4 메인 스레드와 워커 스레드의 구조 . . . . . 8
그림 5 모바일 웹 워커 시스템 구조 . . . . . 9

그림 6 자바스크립트 힙에서의 웹어셈블리 모듈 구조 . . . . . 11
그림 7 메모리 객체와 설정 객체의 복원 코드 . . . . . 12
그림 8 웹어셈블리 함수를 초기화하고 복구한 모듈에 연결시키는 코드 13
그림 9 웹어셈블리를 사용하는 어플리케이션의 복원 과정 . . . . . 15

그림 10 물리 엔진 시뮬레이션 어플리케이션의 구조 . . . . . 17
그림 11 큐브 개수에 따른 스냅샷 복원 시간 . . . . . 19
그림 12 로컬과 연산 오프로딩의 성능 비교 . . . . . 20

# 제 1 장 서론

최근 모바일 기기를 대상으로 하는 복잡한 어플리케이션의 수가 증가하면서 모바일 환경에서 복잡한 연산을 처리하는 것이 요구되고 있다. 이에 따라 자원이 제한된 모바일 환경에서 복잡한 연산을 수행하기 위해 다양한 연구가 진행되고 있다. 이러한 방법 중 하나로써 하드웨어 성능이 제한된 모바일 기기에서 실행중인 어플리케이션의 복잡한 연산을 성능이 좋은 서버에서 대신 수행하는 연산 오프로딩이 제안되었다.

연산 오프로딩을 위해 모바일 기기에서 실행중인 어플리케이션의 연산을 서버에서 이어서 수행하기 위해서는 실행 상태를 저장하여 전송하고 서버에서 복원하는 작업이 필요하다. 이러한 작업을 위해 다양한 방법들이 제안되었으나 상태를 저장하고 복원하기 위한 환경을 구축하기 어렵다는 단점이 있었다[1][11][13][14].

이러한 문제를 해결하기 위해 웹 브라우저와 Node.js[15] 같이 기기에 관계 없이 환경을 구축하기 쉬운 웹 환경의 이식성을 활용하여 웹 어플리케이션을 대상으로 연산 오프로딩을 수행하는 스냅샷 기반의 연산 오프로딩이 제안되었다[2]. 웹 환경의 어플리케이션은 자바스크립트로 구현된다. 스냅샷 기반의 오프로딩은 모바일 기기에서 웹 어플리케이션이 실행되고 있는 상태를 자바스크립트 코드 형태를 가지는 스냅샷[10]으로 저장한다. 서버에서 모바일 기기에서의 실행 상태를 복원하기 위해서는 단순히 스냅샷을 전달받아 웹 환경에서 실행하면 된다. 이러한 특징은 웹 환경의 이식성과 함께 연산 오프로딩을 쉽게 할 수 있도록 만들었다. 그러나 기존의 스냅샷 기반의 연산 오프로딩은 웹 환경이 변화하면서 제안된 새로운 기술인 웹어셈블리[3]에 대해서는 적용되지 못하는 문제가 있었다.

웹어셈블리는 자바스크립트의 부족한 성능을 보완하기 위해 제안된 새로운 형식의 언어이다. C, C++, Rust 등의 정적 타입 언어를 컴파일하여 생성되는 웹어셈블리는 웹 환경에서 네이티브 언어로 구현된 어플리케이션에 가까운 성능을 보여준

다. 웹어셈블리는 웹 어플리케이션의 복잡한 연산을 처리하기 위해 많이 사용되고 있다[7][9].

웹어셈블리는 웹 어플리케이션에서 자바스크립트와는 다른 방식으로 초기화된다. 웹 어플리케이션에서 웹어셈블리를 사용하기 위해서는 프로그램 영역의 역할을 하는 메모리를 할당받고 실행 환경에 맞게 코드를 컴파일하는 과정이 필요하다. 이러한 과정은 기존의 스냅샷 기반 오프로딩에서 웹 어플리케이션의 상태를 저장하고 복원할 수 없도록 만들었다. 사이즈가 큰 메모리는 텍스트 형태로 저장되는 스냅샷에 포함시키기에는 비효율적이다. 또한 컴파일 되는 코드들은 재사용할 수 없고 스냅샷[10]이 저장할 수 없는 네이티브 코드 형태이기 때문에 기존의 스냅샷 기반의 오프로딩 방식을 적용할 수 없다. 웹어셈블리를 사용하여 복잡한 연산을 처리하는 웹 어플리케이션이 증가하면서 문제를 해결하는 것에 대한 중요성이 커지고 있다.

본 논문에서는 웹어셈블리의 초기화 과정을 분석하고 이와 관련된 상태들을 적절하게 복원하는 코드를 생성하여 기존의 스냅샷 기반의 오프로딩이 웹어셈블리를 사용하는 웹 어플리케이션에 대해 적용되지 못하는 문제를 해결하였다. 웹어셈블리를 사용하는 웹 어플리케이션의 오프로딩이 가능해지면서 서버의 리소스를 최대한 활용할 수 있게 되었다.

본 논문의 구성은 다음과 같다. 2장에서는 웹 어플리케이션에서의 웹어셈블리의 배경 지식과 기존의 스냅샷 기반 연산 오프로딩의 문제점 대해서 설명할 것이다. 3장에서는 웹어셈블리를 사용하는 웹 어플리케이션의 상태를 저장하고 복구하기 위한 방법을 설명할 것이다. 4장에서는 웹어셈블리를 사용하는 복잡한 웹 어플리케이션에 제안한 방식을 적용한 결과를 평가할 것이고 5장에서 결론을 이야기할 것이다.

# 제 2 장 배경지식

## 2.1 웹어셈블리

웹 표준 언어인 자바스크립트는 런타임에 컴파일 과정을 수행하는 동적 프로그래밍 언어이다. 자바스크립트는 동적인 특징을 가지고 있어 쉽고 빠른 개발을 할 수 있도록 만들어 주면서 웹 어플리케이션의 개발 언어로 활발하게 사용되고 있다. 현재 자바스크립트로 구현된 어플리케이션의 수행을 빠르게 하기 위해 자바스크립트 엔진이 활발하게 개발되고 있으며 JIT(Just-In-Time) 컴파일이라는 기술이 적용되면서 웹 어플리케이션이 더 빠르게 수행될 수 있게 되었다. 그러나 자바스크립트의 동적인 특징은 문법적으로 통제되어 있는 언어에 비해 제한적인 최적화만을 적용할 수 있어 상대적으로 느린 성능을 보이게 만들었다.

웹어셈블리는 자바스크립트의 부족한 성능을 보완하기 위해 제안된 새로운 형식의 언어이다. 웹어셈블리는 C, C++, Rust 등의 네이티브 언어들을 컴파일하여 생성된다. 그림1은 웹어셈블리의 컴파일 과정을 나타낸다. 웹어셈블리는 실행 환경에 맞게 컴파일 되는 과정만을 남겨둔 상태로 컴파일되기 때문에 어떤 환경에서든 사용할 수 있다는 특징을 가진다. 웹어셈블리의 컴파일 대상 언어들은 변수가 고정된 타입을 가지고 메모리를 메뉴얼하게 관리한다. 따라서 자바스크립트에 비해 상대적으로 많은 최적화를 적용하고 런타임에 GC(Garbage Collection)을 수행할 필요가 없어 네이티브 언어로 구현된 어플리케이션에 가까운 성능을 보여준다. 그러나 웹어셈블리는 웹 플랫폼의 모든 API를 사용할 수는 없기 때문에 실행 도중 자바스크립트로만 실행 가능한 기능을 사용하기 위해서는 초기화 과정에서 자바스크립트 함수를 사용할 수 있도록 참조시키는 과정이 필요하다.

웹어셈블리는 주로 Emscripten[4]이라는 LLVM 기반의 툴체인을 사용하여 컴파일된다. Emscripten을 사용하여 컴파일을 하는 경우 wasm 확장자를 가진 웹어셈

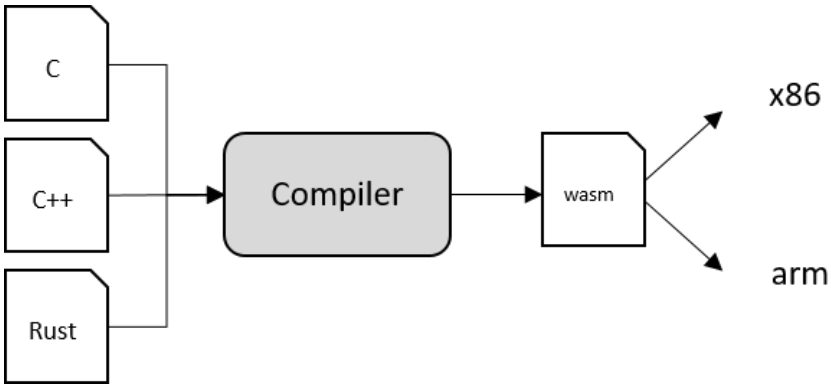



그림 1. 웹어셈블리의 컴파일 과정

블리 파일과 함께 웹어셈블리를 초기화하는 코드가 들어있는 자바스크립트 형태의 접착코드가 생성된다. 웹어플리케이션에서 웹어셈블리를 사용하는 방법으로는 웹어셈블리 파일을 직접 로드하는 방법과 단순히 함께 생성되는 자바스크립트 코드를 실행하는 방법이 있다.

그림 2는 웹 어플리케이션에서의 웹어셈블리 초기화 과정을 나타낸다. 먼저 웹어셈블리 설정 객체 정의, 메모리 할당, 웹어셈블리 파일 읽기 과정을 수행한다. 설정 객체에는 최대 메모리 크기, 상태 변수의 초기값, 웹어셈블리 코드 실행 중 사용할 자바스크립트 함수 등이 포함된다. 웹어셈블리에서 사용하는 메모리는 선형 메모리라고 하며 웹어셈블리의 프로그램 영역의 역할을 수행한다. 선형 메모리는 웹어셈블리와 자바스크립트가 데이터를 전달하는 공간으로 사용되기도 한다. 웹어셈블리 파일을 읽으면 바이트 코드를 읽어온다. 바이트 코드를 사용하기 위해서는 ArrayBuffer에 할당하는 과정을 수행해야 한다. 이후 앞서 생성한 메모리, 설정 객체, ArrayBuffer를 사용하여 컴파일 과정을 수행하고 자바스크립트 객체 형태의 모듈을 생성한다. 모듈 내에는 컴파일 된 웹어셈블리 함수와 메모리, 그리고 모듈의 상태 정보가 포함되어있다.

웹어셈블리를 통해 웹 환경에서 복잡한 연산을 처리할 수 있게 되면서 자바스크

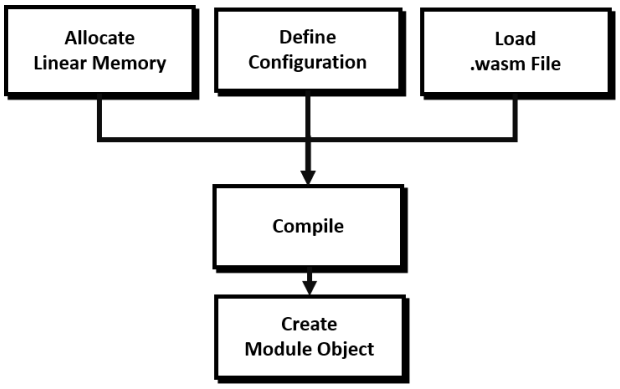



그림 2. 웹 어플리케이션에서의 웹어셈블리 초기화 과정

립트로 실행되던 복잡한 연산 부분을 대체하거나 컴퓨터 비전, 영상 처리, 머신 러닝 등 다른 언어로 구현되었던 다양한 라이브러리들이 웹어셈블리의 형태로 배포되기 시작했고 그 중요성은 점점 커지고 있다[7][9].

## 2.2 스냅샷 기반 연산 오프로딩

스냅샷[10]은 어떤 시점의 프로그램의 실행 상태를 저장한 것을 의미한다. 웹 어플리케이션의 상태는 자바스크립트의 형태로 저장할 수 있다. 웹 어플리케이션의 실행 상태는 화면에 표시되는 요소들의 상태를 나타내는 DOM(Document Object Model) tree와 자바스크립트의 상태로 나타낼 수 있다.

DOM은 HTML 문서의 프로그래밍 인터페이스로, 구조화된 표현을 통해 자바스크립트에서 접근하여 정적인 웹 페이지를 통제할 수 있도록 도와준다. DOM 객체들은 트리 형태의 구조로 파싱되는데 이를 DOM tree라고 한다. 브라우저의 랜더링 엔진에서는 이 정보를 읽고 랜더링을 수행한다. 자바스크립트의 상태는 이벤트 핸들러와 이벤트들에 대한 정보, 그리고 객체 등의 실행 상태로 구성된다.

웹 어플리케이션의 상태 대한 정보는 브라우저의 루트 객체인 window에 포함

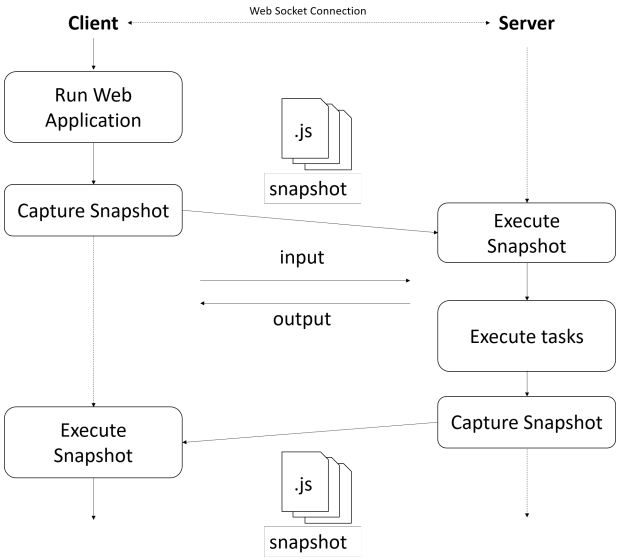



그림 3. 스냅샷 기반 연산 오프로딩의 실행 과정

되어 있다. 스냅샷은 window 객체를 탐색하여 DOM tree와 자바스크립트의 실행 상태에 대한 정보를 포함하는 자바스크립트 코드로 생성된다. 이벤트와 함수 안에서 다른 scope에서 참조되는 클로저에 대한 정보는 일반적인 방법으로는 구할 수 없기 때문에 브라우저 내부의 정보를 읽어서 가져온다. 생성된 스냅샷을 단순히 다른 기기의 브라우저나 Node.js에서 실행하기만 하면 이전 환경에서의 실행 상태로 복구된다. 스냅샷 기반 연산 오프로딩은 스냅샷의 장점을 활용하는 연산 오프로딩 방식이다. 저사양 기기에서 실행중인 웹 어플리케이션의 상태를 저장한 스냅샷을 성능이 좋은 서버에서 복구하여 연산을 대신 수행할 수 있도록 만들어준다.

그림 3은 스냅샷 기반 연산 오프로딩의 수행 과정을 나타낸다. 먼저 저사양 모바일 기기에서 웹 어플리케이션을 실행한다. 그 후 연산 오프로딩을 원하는 시점에 window 객체를 탐색하여 스냅샷을 생성한다. 이 때 클라이언트와 서버가 연결되어 있는지 확인하고 연결되지 않았다면 연결을 시도한다. 클라이언트와 서버의 연결이

확인되면 클라이언트는 생성한 스냅샷을 서버로 전송한다. 서버는 전송된 스냅샷을 실행하여 웹 어플리케이션 실행 상태를 복원한다. 이 후 서버에서는 클라이언트로부터 입력값을 받아 연산을 대신 수행하고 결과값을 돌려준다. 연산 오프로딩을 끝내고 클라이언트에서 수행을 이어가고 싶은 경우에는 서버의 실행 상태를 스냅샷으로 저장하여 클라이언트로 전송하기만 하면 된다.

그러나 기존의 스냅샷 기법은 자바스크립트와는 다른 방식으로 생성되는 웹어셈블리의 상태를 저장하고 복원하지 못한다는 문제가 있다. 웹어셈블리 함수들은 초기화 하는 과정에서 실행 환경에 맞게 컴파일 하는 과정을 수행한다. 컴파일 된 웹어셈블리 함수는 코드로 저장할 수 있는 자바스크립트 함수의 형태가 아닌 네이티브 코드의 형태를 가진다. 웹어셈블리 함수와 마찬가지로 네이티브 함수의 형태인 built-in API의 경우 웹 환경이 갖추어지면 어디서든 사용할 수 있어 같은 함수를 실행하는 코드를 생성하는 방식으로 해결할 수 있었다. 그러나 웹어셈블리 함수는 컴파일을 통해 생성되기 때문에 built-in API와 같은 방식으로 해결할 수 없다. 웹어셈블리의 메모리의 경우 최소 16MB 이상의 사이즈를 가진다. 텍스트 형식으로 저장하는 기존의 스냅샷 방식으로 저장하게 된다면 용량이 커지게 되면서 저장 및 복원의 성능을 저하시키는 원인이 된다. 최근 웹어셈블리를 통해 복잡한 연산을 처리하는 사례가 많아지게 되면서 이 문제를 해결하는 것의 중요성이 커지고있다.

## 2.3 모바일 웹 워커

웹 워커[12]는 웹 어플리케이션의 백그라운드 스레드에서 스크립트를 실행할 수 있도록 만들어주는 기술이다. 웹 워커는 사용자 인터페이스에 영향을 주는 메인 스레드를 방해하지 않고 작업을 수행하기 위해 사용되며 주로 네트워크나 I/O 작업, 연산량이 많은 작업을 수행한다.

웹 워커는 메인 스레드로부터 수행할 자바스크립트 코드를 전달받아 생성된다. 웹 워커가 생성되면 새로운 스레드가 생성되고 자신만의 분리된 영역에서 전달받

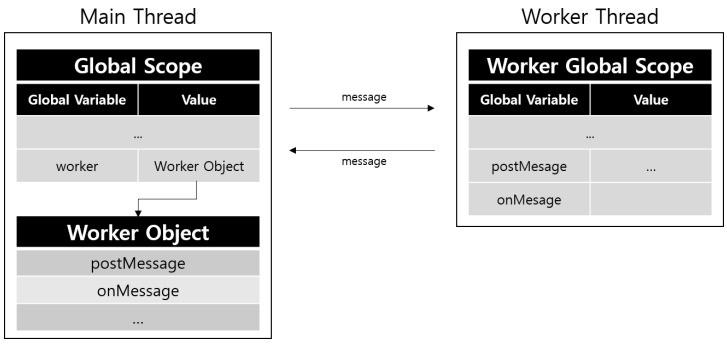



그림 4. 메인 스레드와 워커 스레드의 구조

은 자바스크립트 코드를 실행한다. 이 때 메인 스레드와 워커 스레드 사이의 메시지 통신을 위한 채널이 생성된다. 이 후 메인 스레드와 워커 스레드는 메시지 이벤트를 통해 데이터를 통신하고 이벤트를 실행한다.

그림 4는 웹 워커가 생성되고 나서 메인 스레드와 워커 스레드의 구조를 나타낸다. 먼저 메인 스레드에서는 웹 워커에서 수행할 소스 코드를 매개변수로 주고 웹 워커 객체를 생성한다. 생성된 웹 워커는 전달받은 소스코드를 실행한다. 이 때 웹 워커는 메인 스레드와 다른 독립적인 스코프를 가진다. 웹 워커가 생성되면 메인 스레드와 워커 스레드 사이의 메시지 채널이 생성되며 이 채널을 통한 통신은 다른 스레드로 메시지를 보내는 postMessage와 메시지를 받았을 때 실행되는 이벤트인 onMessage를 통해 이루어진다.

모바일 웹 워커는 웹 워커를 대상으로 하는 스냅샷 기반 연산 오프로딩 기술이다. 웹 어플리케이션 전체가 아닌 웹 워커의 스레드만 저장하고 서버에서 복구시키기 때문에 더 가볍고 유연하다는 특징을 가진다. 모바일 웹 워커의 시스템은 웹 소켓을 통해 데이터를 통신하는 서버 프로세스와 웹 워커의 상태를 복원하고 연산을 수행하는 프로세스로 구성된다.

그림 5는 모바일 웹 워커 시스템의 구조를 나타낸다. 모바일 웹 워커는 웹 워커

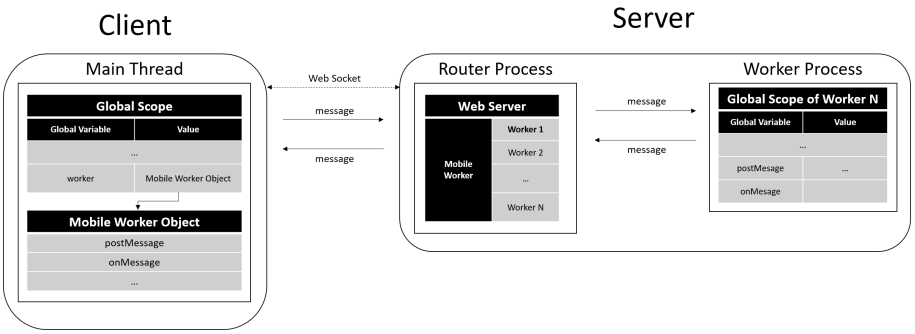



그림 5. 모바일 웹 워커 시스템 구조

를 생성할 때 웹 워커 객체 대신 모바일 웹 워커 객체를 생성한다. 모바일 웹 워커 객체는 기존의 웹 워커 객체가 스냅샷을 생성하고 서버와 통신할 수 있는 기능을 포함하고 있다. 모바일 웹 워커는 연산 오프로딩을 수행하기 전에는 기존의 웹 워커와 같은 역할을 수행하며 연산 오프로딩을 수행하는 경우 웹 서버와 웹 소켓을 생성한 다음 오프로딩을 수행할 웹 워커의 스냅샷을 생성하고 서버로 전송한다. 웹 서버 프로세스에서는 클라이언트로부터 스냅샷을 전달받으면 웹 워커의 역할을 수행할 프로세스를 생성하고 스냅샷을 전달한다. 워커 프로세스는 전달받은 스냅샷을 사용하여 상태를 복원한다. 이 후 클라이언트의 메인 스레드와 서버에 생성된 모바일 워커는 postMessage, onMessage를 사용하여 통신을 한다. 이 때 메시지는 서버 프로세스를 거치게 되며 서버 프로세스는 받은 메시지를 적절한 위치로 데이터를 전달하는 역할을 수행한다.

메인 스레드 대신 웹 워커를 연산 오프로딩 시키는 모바일 워커는 동시에 여러 개의 웹 워커를 오프로딩 시킬 수 있다. 이를 통해 웹 워커를 선택적으로 오프로딩 시키거나 여러개의 서버에 분산해서 오프로딩 시킴으로써 동시에 여러 서버의 자원을 활용할 수 있다는 장점이 있다.

# 제 3 장 웹어셈블리 상태 저장 및 복원

웹어셈블리는 C, C++, Rust 등의 정적 타입 언어를 컴파일하여 생성되는 코드로 웹 환경에서 자바스크립트와는 다른 방식을 거쳐 초기화된다. 웹 어플리케이션에서 웹어셈블리를 초기화하면 객체 형태의 웹어셈블리 모듈이 생성된다. 그림 6은 자바스크립트 힙 메모리 내에서의 웹어셈블리 모듈의 구조를 나타낸다. 웹어셈블리의 모듈은 선형 메모리의 상태 변수들, 웹어셈블리 함수들, 그리고 웹어셈블리의 프로그램 실행 상태가 저장되는 선형 메모리로 구분된다. 상태 관련 변수들은 선형 메모리에서 필요한 오프셋과 크기 정보를 저장한다. 웹어셈블리 함수들은 컴파일 과정에서 환경에 맞게 컴파일된 함수들이며 자바크스립트 함수의 형태가 아닌 컴파일 된 형태를 가진다. 선형 메모리는 스택 영역, 글로벌 영역, 힙 영역의 역할을 수행하며 웹어셈블리의 실행 상태가 저장된다. 선형 메모리는 최소 16MB의 메모리를 할당받고 부족한 경우 추가적으로 할당받을 수 있다. 자바스크립트와 웹어셈블리가 데이터를 전달하는 공간으로 사용하기도 하며 배열과 같은 복잡한 자료 구조를 가진 데이터를 전달하기 위해서는 선형 메모리에 데이터를 할당하고 데이터가 저장된 위치를 가리키는 오프셋을 사용해야 한다.

웹어셈블리의 상태를 복원하기 위해서는 웹어셈블리 상태 변수들과 함수들, 그리고 선형 메모리를 모두 복구해야 한다. 상태 변수들의 경우 일반적인 자바스크립트 객체의 프로퍼티와 같기 때문에 저장이 가능하다. 그러나 웹어셈블리 함수는 코드로 저장할 수 있는 자바스크립트 합수의 형태가 아니고 컴파일하는 과정의 결과로 생성되는 네이티브 함수이기 때문에 저장이 불가능하다. 선형 메모리의 경우 사이즈가 크기 때문에 텍스트 형태로 저장하는 스냅샷에 포함시키기에는 비효율적이다. 웹어셈블리의 상태를 복원하기 위해서는 웹어셈블리 함수를 초기화하는 코드를 생성하고 선형 메모리를 스냅샷이 아닌 다른 방식으로 처리해주는 과정이 필요하다.

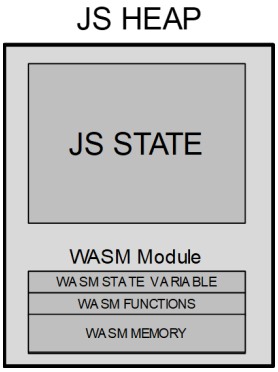



그림 6. 자바스크립트 힙에서의 웹어셈블리 모듈 구조

## 3.1 웹어셈블리 함수의 복원

웹어셈블리 함수는 실행 환경에 의존적이기 때문에 초기화 과정에서 실행 환경에 맞게 컴파일된다. 웹어셈블리 함수를 복원하기 위해서는 새로운 환경에서 다시 컴파일 과정을 수행해야 한다. 이를 위해서는 스냅샷을 생성할 때 웹어셈블리 함수를 제외시키고 새로운 환경에서 컴파일 된 함수를 복원된 모듈 객체에서 참조하도록 만들어야 한다. 서버에서 접착 코드를 실행하는 방식을 사용할 수도 있지만 새로운 모듈을 생성하면서 불필요한 과정까지 포함되기 때문에 비효율적이다. 효율적으로 스냅샷을 복원시키기 위해서는 웹어셈블리 함수만을 초기화하여 복원된 객체에 참조시키는 방법이 필요하다.

웹어셈블리 함수를 초기화하는 과정에서 웹어셈블리 메모리 객체와 설정 객체를 사용한다. 메모리 객체는 접착 코드에 정의된 값을 사용하여 생성되고 설정 객체는 접착 코드에 정의된 값과 함수를 포함하고 있다. 접착코드를 사용하면 비효율적으로 복구되기 때문에 접착코드를 사용하지 않고 이 객체들에 대한 정보를 얻기 위한 다른 방법이 필요하다. 이러한 객체들에 대한 정보는 웹어셈블리 모듈에 포함되어 있거나 클로저의 형태로 숨겨져있다. 이 문제를 해결하기 위해 스냅샷을 생성

```
1  var obj_ref = new Array();
2  // 1. Fill in the values in the reference table
3  // ...
4  // 2. restore JavaScript state
5  // ...
6
7  // 3. Allocate memory for WebAssembly Module
8  Module['wasmMemory'] = new WebAssembly.Memory({'initial' : 33554432 / 65536, 'maximum': 1024 });
9  // Size of memory uses the value obtained during the snapshot capture
10
11 // 4. Create views and fill in the empty value in the reference table
12 obj_ref[0] = Module['buffer'] = Module['wasmMemory'].buffer;
13 obj_ref[1] = Module['HEAP8'] = new Int8Array(Module['buffer']);
14 obj_ref[2] = Module['HEAP16'] = new Int16Array(Module['buffer']);
15 // ...
16
17 // 5. Define Import Object
18 let importObject = {};
19 importObject['env'] =  Module['asmLibraryArg']; // get value from recovered JavaScript state
20 importObject['asm2wasm'] = {
21     "f64-rem": function(x, y) { return x % y },
22     "debugger": function() { debugger; }
23 };  // generated code using state information obtained during snapshot capture
24 importObject['memory'] = Module['wasmMemory']; // use newly alloacted memory
25 // ...
```

그림 7. 메모리 객체와 설정 객체의 복원 코드

하는 과정에서 메모리 객체와 설정 객체를 초기화할 때 필요한 값에 대한 정보를 저장하고 웹어셈블리 모듈이 복원되고 난 후 저장한 값과 모듈에 복원된 정보를 사용하여 메모리 객체와 설정 객체를 복원하는 코드를 스냅샷에 포함시키는 방법을 고안하였다.

그림 7은 메모리 객체와 설정 객체를 복원하기 위해 생성된 코드를 나타낸다. 복구 코드는 스냅샷을 생성과정에서 저장한 웹어셈블리와 관련된 프로퍼티의 값을 토대로 생성된다. 먼저 복원 가능한 자바스크립트의 상태를 복원한다(line 1-5). 스냅샷을 효율적으로 생성하기 위해 복원된 자바스크립트 상태로부터 불러올 수 있는 값은 해당 값을 불러오는 코드로 대체했다. 이 후 웹어셈블리 메모리를 초기화 하고 해당 메모리에 대한 뷰를 생성한다.(line 7-14). 마지막으로 웹어셈블리 설정 객체를 복원한다(line 17-25). 이 때 line 19처럼 복원된 자바스크립트 상태로부터 값을 불러오거나 line 20-23 처럼 값을 초기화해주는 코드를 실행한다.

```
1   // 6. Read WebAssembly File
2   var source = readFileSync('module.wasm');
3   var arrayBuffer = new Uint8Array(source);
4
5   // 7. Instantiate WebAssembly function with recovered import object
6   WebAssembly.instantiate(arrayBuffer, importObject)
7              .then((result) => {
8                 // 8. Fill in the empty value in the reference table
9                 obj_ref[1001] = result.instance.exports['stackSave'];
10                obj_ref[1002] = result.instance.exports['stackRestore'];
11                // ...
12             })
```

그림 8. 웹어셈블리 함수를 초기화하고 복구한 모듈에 연결시키는 코드

메모리 객체와 설정 객체를 복구하면 웹어셈블리 파일을 읽고 인스턴스화를 하는 과정을 수행하면 웹어셈블리 함수를 컴파일 할 수 있다. 이 과정을 수행하기 위해 클라이언트에서는 웹어셈블리 파일을 함께 전송해야 한다. 스냅샷을 생성하는 과정에서 웹어셈블리 함수를 제외했기 때문에 복구한 모듈이 새롭게 초기화한 함수들을 참조할 수 있도록 연결하는 과정이 필요하다. 그림 8은 웹어셈블리 함수를 초기화하고 복구한 모듈에 연결시키기 위해 생성된 코드를 나타낸다. 브라우저와 Node.js는 상당 부분 호환이 가능하지만 일부 API는 지원되지 않는다. 웹어셈블리 파일을 읽을 때 브라우저에서는 fetch API를 사용하고 Node.js 에서는 파일 시스템을 사용한다. 이러한 점을 고려하여 서버의 Node.js의 환경을 지원하기 위해 파일 시스템을 통해 웹어셈블리를 읽는 코드를 생성했다(line 1-3). 웹어셈블리 코드를 읽으면 ArrayBuffer에 할당하고 앞서 복구한 메모리 객체와 설정 객체를 사용하여 컴파일을 수행한다(line 5). 이 후 생성된 함수들을 모듈에 연결시키는 코드를 생성하여 정상적으로 복구될 수 있도록 만들었다(line 8-11).

## 3.2 웹어셈블리 메모리의 복원

접착 코드로 생성되는 웹어셈블리 메모리의 최소 사이즈는 16MB 이다. 메모리의 사이즈는 공간이 부족한 경우 추가로 할당받을 수 있다. 메모리는 전역 변수, 스택, 힙 데이터로 이루어진 웹어셈블리의 모든 상태를 포함하고 있다. 웹어셈블리 메모리의 경우 기존의 스냅샷 기법을 사용하여 상태를 저장할 수 있다. 그러나 텍스트 형태로 저장하는 기존의 스냅샷 기법으로 이진 형태의 데이터를 저장하고 전송하게 되면 사이즈가 커지기 때문에 비효율적이다. 이러한 문제를 해결하기 위해 웹어셈블리 메모리의 데이터를 스냅샷에 포함시키지 않고 별도로 전송하는 방법을 사용하였다. 서버에서는 스냅샷과 웹어셈블리 메모리를 동시에 전송받게 되며 전송받은 웹어셈블리 메모리의 데이터는 메모리 복원이 끝난 후 할당된다. 스냅샷의 크기가 작아지게 되면서 더 이른 시점에 스냅샷을 전송받아 메모리가 필요한 부분을 제외한 복원 과정을 미리 수행할 수 있다. 이 후 메모리 데이터를 전송 받으면 나머지 과정을 수행하게 된다. 여러 복원 과정을 동시에 수행하게 되면서 스냅샷을 더 효율적으로 복원할 수 있게되었다. 그림 9는 새로운 스냅샷 기반의 오프로딩 방식의 수행 과정을 나타낸다.

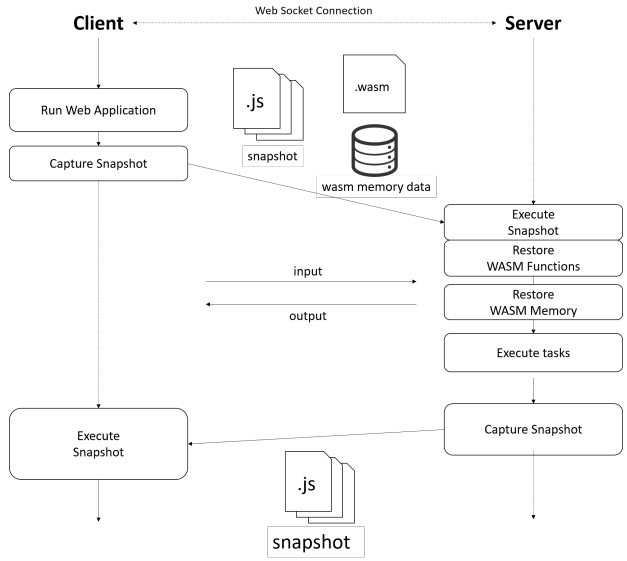



그림 9. 웹어셈블리를 사용하는 어플리케이션의 복원 과정

# 제 4 장 실험 및 결과

## 4.1 실험 환경

실험은 클라이언트로 ARM Cortex-A15 2GHz, 2GByte RAM의 사양인 Odroid XU4[6] 보드와 서버로 Intel i7 -7700 CPU 3.60GHz 32GByte RAM의 사양의 데스크탑 환경에서 수행하였다. 운영체제는 모두 16.04.3 LTS 버전을 사용하였다. 네트워크는 업로드 10Mbps, 다운로드 36Mps 환경[8]에서 수행하였다.

실험에는 웹어셈블리로 포팅된 물리 엔진 시뮬레이션 어플리케이션을 사용하였다[7]. 수백개 이상의 큐브들에 작용하는 중력이나 관성 충돌 등의 물리적 현상을 시뮬레이션 하는 어플리케이션으로 많은 연산량을 필요로 한다. 그림 10은 물리 엔진 시뮬레이션 앱의 구조를 나타낸다. 웹 어플리케이션은 메인 스레드의 연산이 많으면 병목현상이 생기기 때문에 부담을 줄여주기 위해 웹 워커를 생성하여 연산량이 많은 작업인 시뮬레이션을 수행하도록 만들었다. 메인 스레드에서 물체의 정보를 저장하는 객체를 생성하고 초기 위치를 주면 웹 워커에서 물체들간의 물리 법칙을 시뮬레이션 한다. 이 때 물리 엔진은 웹어셈블리로 구현된 함수를 사용한다. 시뮬레이션 과정이 끝나면 새로운 위치를 반환하고 메인 스레드에서는 변경 사항을 반영하고 랜더링을 수행한다.

연산 오프로딩 방식으로는 모바일 웹 워커 시스템을 사용하였다. 새롭게 제안된 스냅샷 기법의 활용성에 대해 알아보기 위해 스냅샷의 저장 시간과 복구 시간, 크기에 대해 측정하였다. 그리고 웹어셈블리를 사용하는 웹 어플리케이션의 복원이 가능해지면서 얻을 수 있는 성능이득에 대해서 알아보기 위해 수행 시간과 초당 프레임 수를 측정하여 성능을 비교하였다.

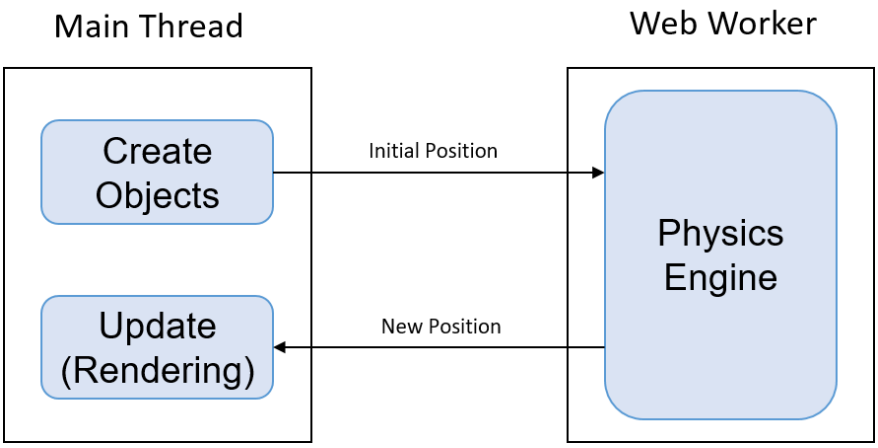



그림 10. 물리 엔진 시뮬레이션 어플리케이션의 구조

## 4.2 스냅샷 기반의 상태 저장 및 복원 성능

새롭게 제안된 스냅샷의 활용성에 대해 알아보기 위해 큐브의 개수에 따른 스냅샷과 웹어셈블리 메모리의 크기 변화와 복원 시간에 대해 측정을 진행하였다.

표 1은 큐브의 개수에 따른 스냅샷과 웹어셈블리 메모리의 크기를 나타낸다. 큐브의 수가 늘어나는 것은 자바스크립트의 큐브 객체의 수가 늘어나는 것을 의미한다. 따라서 스냅샷을 생성할 때 탐색하고 저장해야하는 객체의 수가 늘어나게 되면서 스냅샷의 크기와 캡처 시간이 증가하는 것을 확인할 수 있다. 웹어셈블리 모듈을 전부 탐색하지 않고 스냅샷을 통해 복원되지 못하는 웹어셈블리 함수와 메모리는 제외시키고 코드를 생성한다는 점에서 기존 스냅샷과 차이가 있다. 코드의 생성은 탐색을 할 때 얻은 정보를 사용하여 이루어지기 때문에 전체 캡처 시간에서 작은 부분만을 차지한다. 웹어셈블리 메모리의 크기의 경우 일정 개수까지는 유지되다가 증가하는 것을 확인할 수 있다. 이것은 할당받은 메모리가 충분하지 않아 추가적으로 메모리를 할당받은 것을 의미한다. 웹어셈블리 메모리는 스냅샷과 별개로 관리되기 때문에 스냅샷의 캡처 시간에서는 제외된다.

표 1. 큐브 개수에 따른 스냅샷 캡처 시간 및 스냅샷과 웹어셈블리 메모리의 크기

| Cubes | Snapshot Capture Time (ms) | Snapshot Size (KB) | WASM Memory Size (MB) |
|---|---|---|---|
| 50 | 1059 | 2221 | 16 |
| 100 | 1153 | 2300 | 16 |
| 200 | 1285 | 2459 | 16 |
| 500 | 1527 | 2937 | 16 |
| 1000 | 2102 | 3751 | 32 |

그림 11은 큐브 개수에 따른 마이그레이션 시간을 나타낸다. 실험 결과는 스냅샷 캡처 시간, 상태 복구 시간, 그 외 시간으로 나누어서 측정하였다. 상태 복구 시간은 큐브의 개수에 따라 392ms 부터 792ms까지 증가하는 것을 확인할 수 있다. 스냅샷의 크기가 증가하면서 실행해야하는 코드의 수도 증가하게 되면서 시간이 늘어나게 된 것을 알 수 있다. 그 외 시간에는 메모리 데이터 전송 대기 시간, 모바일 워커 시스템의 오버헤드가 포함된다. 표 1과 그림 11의 실험 결과를 통해 새롭게 제안된 스냅샷의 저장과 복원이 수 초 안에 이루어 질 수 있고 적은 양의 데이터 전송을 필요로 하는 것을 확인할 수 있다. 이것은 새로운 스냅샷 기반의 연산 오프로딩 방식이 다른 가상머신 및 컨테이너 기반의 접근 방식보다 여전히 가볍고 빠르다는 것을 보여준다. 또한 미래에 5G 인터넷 인프라가 구축된다면 네트워크에 의한 지연 시간이 줄어들게 되면서 더 빠른 시간 안에 마이그레이션이 이루어 지는 것을 기대할 수 있다.

## 4.3 실행 성능

연산 오프로딩을 통해 얻을 수 있는 성능 향상을 알아보기 위해 실행 시간을 측정했다. 클라이언트와 서버의 순수한 실행 시간을 비교하기 위한 시뮬레이션 시간과

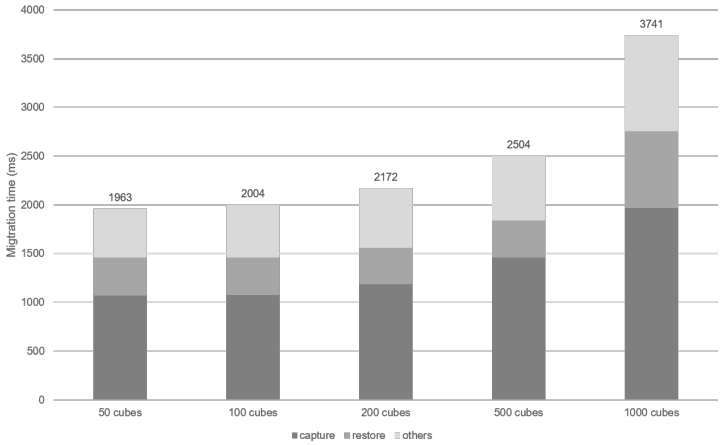
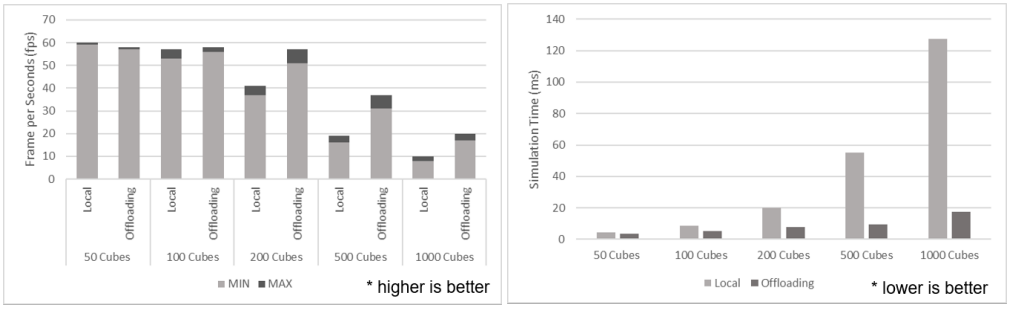



그림 11. 큐브 개수에 따른 스냅샷 복원 시간

서버로부터 시뮬레이션 결과 전달받고 랜더링을 수행하는 시간을 알아보기 위해 초당 프레임 수를 측정하였다.

그림 12는 초당 프레임 수와 시뮬레이션 시간에 대해서 측정을 한 결과를 나타낸다. 서버에서 연산 오프로딩을 수행하는 경우 서버의 실행시간은 최소 1.25배 에서 큐브의 수가 증가함에 따라 최대 5.88배까지 증가하는 것을 확인할 수 있다. 이것은 큐브의 수가 증가할수록 연산이 복잡해지면서 클라이언트의 제한된 자원으로는 감당할 수 없게 되는 것을 알 수 있다. 초당 프레임 수는 큐브의 수가 적을 때는 연산 오프로딩을 수행하지 않은 것이 약간 더 좋은 성능을 보여주다가 일정 개수가 넘어가면서 연산 오프로딩의 성능이 더 좋아지는 것을 확인할 수 있다. 클라이언트의 자원으로도 충분히 감당할 수 있는 연산량에서는 연산 오프로딩을 통해 얻을 수 있는 이득이 적고, 오히려 네트워크의 부담이 더해지면서 약간의 성능 저하가 발생한 것으로 보인다. 그러나 큐브의 수가 증가할수록 연산의 복잡도가 증가함에 따라 실행 시간의 차이가 증가하고 클라이언트의 자원으로는 원활하게 처리할 수 없게 되면서 연산 오프로딩을 사용한 것이 더 좋은 초당 프레임수를 보여주게 된 것으로




(a) 초당 프레임 수 (b) 시뮬레이션 실행 시간

그림 12. 로컬과 연산 오프로딩의 성능 비교

보인다.

초당 프레임 수의 결과를 보면 연산 오프로딩을 수행하는 경우 시뮬레이션 시간이 굉장히 짧은데도 불구하고 초당 프레임 수가 줄어드는 것을 확인할 수 있다. 이것은 큐브의 수가 증가하면서 연산에 대한 부담 뿐만 아니라 랜더링에 대한 부담도 증가하면서 랜더링 성능에 의해 초당 프레임수가 제한되었기 때문이다. 오프로딩을 통해 빠른 수행이 가능한데도 불구하고 클라이언트의 랜더링 성능에 의해 병목이 발생하면서 기대치보다 낮은 성능을 보여주는 것을 확인할 수 있다. 1000개 이상의 객체에 대한 수행 결과를 보면 연산 오프로딩을 수행할 때와 로컬에서 수행할 때 느리게 만드는 주요한 원인이 서버는 랜더링 시간, 클라이언트는 시뮬레이션 시간으로 서로 다른 것을 확인할 수 있다. 이러한 결과는 연산 오프로딩이 클라이언트의 부족한 성능을 보완주고 성능 저하의 원인을 줄여주는 역할을 할수 있다는 것을 보여준다.

# 제 5 장 결론

기존의 스냅샷 기반의 연산 오프로딩은 웹어셈블리를 사용하는 웹 어플리케이션에 적용되지 못하는 문제가 있었다. 컴파일 과정을 거쳐 실행 환경에 맞게 생성되는 웹어셈블리 함수와 샌드박싱을 위해 할당된 큰 사이즈의 메모리는 스냅샷을 통해 웹어셈블리의 상태를 저장하고 복원할 수 없었다.

본 논문에서는 웹어셈블리의 실행 상태를 복원하기 위해 웹어셈블리 함수를 초기화 하는 코드를 생성하고 메모리를 별도로 관리하는 방식을 제안하여 웹어셈블리를 사용하는 어플리케이션에 대해 스냅샷 기반의 오프로딩을 할 수 있도록 만들었다.

웹어셈블리는 웹에서 복잡한 어플리케이션을 실행하기 위해 사용되고 있으며 그 중요성은 더욱 커지고 있다. 웹어셈블리를 사용하는 웹 어플리케이션에 대해서 스냅샷 기반의 연산 오프로딩이 가능하게 되면서 성능이 뛰어난 서버에서 웹어셈블리를 사용함으로써 서버의 자원을 최대한 활용할 수 있게 되었다.

# 참고 문헌


[1] Chun, Byung-Gon, et al. ”Clonecloud: elastic execution between mobile device and cloud.” Proceedings of the sixth conference on Computer systems. ACM, 2011.

[2] Jeong, Hyuk-Jin, and Soo-Mook Moon. ”Offloading of web application computations: A snapshot-based approach.” 2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing. IEEE, 2015.

[3] Haas, Andreas, et al. ”Bringing the web up to speed with WebAssembly.” ACM SIGPLAN Notices. Vol. 52. No. 6. ACM, 2017.

[4] https://emscripten.org/index.html

[5] Deutsch, Peter. DEFLATE compressed data format specification version 1.3. No. RFC 1951. 1996.

[6] https://www.hardkernel.com/ko/

[7] https://github.com/kripken/ammo.js/

[8] S. G. Index. Average internet speed of united states, 2019.

[9] https://docs.opencv.org/3.4/index.html

[10] Oh, JinSeok, et al. ”Migration of web applications with seamless execution.” ACM SIGPLAN Notices. Vol. 50. No. 7. ACM, 2015.

[11] Ha, Kiryong, et al. ”Just-in-time provisioning for cyber foraging.” Proceeding of the 11th annual international conference on Mobile systems, applications, and services. ACM, 2013.

[12] https://html.spec.whatwg.org/#workers

[13] Cuervo, Eduardo, et al. ”MAUI: making smartphones last longer with code offload.” Proceedings of the 8th international conference on Mobile systems, applications, and services. ACM, 2010.

[14] Satyanarayanan, Mahadev, et al. ”The case for vm-based cloudlets in mobile computing.” IEEE pervasive Computing 4 (2009): 14-23.

[15] https://nodejs.org/ko/

# ABSTRACT

## Snapshot-based Computation Offloading for Web Application Using WebAssembly

ChangHyun Shin
Dept. of Electrical and Computer Engineering
The Graduate School
Seoul National University

Recently, methods for effectively executing an application on a mobile device lacking hardware performance are being studied. Computation offloading is one of the trials that migrating computations from resource-constrained mobile device to server. To perform computaion offloading is difficult, because it is necessary to store the execution state of application, transmit it, and restore it in the server. Snapshot are technique that save and restore the execution state of a web application. However, existing snapshot-based computation offloading methods do not support WebAssembly.

In this paper we propose new snapshot based computation offloading which can save and restore WebAssebmly state. To restore the state of WebAssembly, we generated code and captured memory to restore WebAssembly state. We evaluate out approach by applying Web Application using WebAssembly. The result shows our method performs migration within a shot time and increase the execution performance.

**keyworkds**: WebAssembly, offloading, snapshot, JavaScript, web

**Student Number**: 2017-28910